Notes on Laning's 1953 Whirlwind program.

D. Knuth 12 Aug 76 - 14 Aug 76

Whirlwind opcodes (as deduced from this program, not having a manual available at this time) 16-bit words; 5 bits for the opcode.

- ad — ac ← ac + (m). Set toggle iff result ≤ 0 or something like this. [-1+1 sets it; 0-0 does too].
- ao — equiv to ca m, ad=1=, ts m.
- ca — ac ← (m). toggle apparently not set? see 15a1; but yes set: see 3a15.
- cl — stop
- cp — jump if toggle.
- cs — ac ← −(m) also seems to set toggle, see 21a13
- dv — divide [used for interpreter call only]
- ex — ac ⟷ (m)
- mr — multiply [used for interpreter call only]
- p — decimal integer constant. (ac←op.)
- rd — read a character in Flexowriter code into ac. [6-bit code]
- rc
- sf — normalize ac and store number of shifts in (m) [I guess; used only once, 2Ya1]
- si — unknown; something about I/O?
- sl — shift left
- sp — jump and set J register to address of following instruction
- sr* — shift right into extension of accumulator
- su — ac ← ac − (m), toggle iff result ≤ 0
- ta — address(m) ← J register.
- td — address(m) ← address(ac)
- ts — (m) ← ac

Assembly language: 4a3 means location of a3 plus 4; 2r means location of current routine plus 2; octal constants 0.77410. Location 0 contains 0.

There is a 128-place switching table (unused positions are filled with miscellaneous constants and snatches of code), divided between lower case and upper case. The codes are:

- a, b, .., z → letter [50a4]
- 0, 1, .., 9 → 1, 2, .., 10
- ' , ..., 9 → 11, 12, .., 20
- space, backspace, tab, cr, stop, null, • → next [2a1].
- = → equals [10a10] + → plus [52a4] − → minus [55a4] / → divide [0a9] (→ print [0a11]) → upprint [0a12]
- . → 0 or period [8a2], a switch initially set 0
- , → comma [14a10] or next or spcomma, a switch initially set to comma
- uc → upper [42a4] lc → lower [0a4] S → ucs [40a10] C → ucc [50a10] → expminus [10a9].
- P → print [0a2] or print2 [2a2] a switch initially set to print 2

The code uses switched instructions, I'll use Boolean flags all initially false

- cflag: upper case?
- nflag: processing a constant? if so, z = number [0a5] or exponent [0a6]
- nflag: constant is not a label
- pflag: decimal point occurred?
- tflag: implied multiplication in effect rather than mnemonic label. actually, tflag = nflag, so drop it
- rflag: processing a negative exponent
- eflag: exponent should be negative (or positive if in the denominator)

mem[ ] = memory loc j (shared by compiler and object code and interpretive routines) opart[ ] = opcode field of mem[ ]
addrpart[ ] = address field of mem[ ].
flac = floating point accumulator [initially 0.0]. digits, rdigits = counters initially zero.
addr = next avail prog address [initially 32] caddr = last address used for constants [initially 197].
op = operation code initially "mr"; op1 initially "dv" [always the opposite of op]
k = parenthesis level

```
0a1  start:   "si 128";   whatever that means
2a1  next:    c ← read next char; if cflag then x ← uppercase code[c] else x ← lowercase code[c];
9a1           if x is a label then (if nflag then (nflag ← false; go to z) else go to x);
12a1          nflag ← nflag; if x=0 [code for period] then (pflag ← true; go to next);
17a1          if x ≤ 10 then (z ← number; d ← x-1) else (z ← exponent; d ← x-11);
23a1          fd ⇐ float(d); flac ⇐ flac ⊗ 10.0 ⊕ fd;
36a1          if pflag then rdigits ← rdigits+1 else ldigits ← ldigits+1;
37a1          go to next;
50a4  letter: if nflag then (compile(op, c); go to next);
28a10 label:  ~~if eflag then false~~ nflag ← true;
32a10 [19a6]  n ← decode;                    decode = return (unfloat ( if eflag then -flac else flac)
33a10         mem[switchtable + n] ← ("sp", addr);    comment n=0 not allowed, since 0 always implied as label;
9a6   reset:  flac ⇐ 0.0; ldigits ← rdigits ← 0; pflag ← false;
17a6          if rflag then (eflag ← ~eflag; rflag ← false);
38a4          go to x;
10a10 [0a10]  equals: l ← eqsub;    eqsub = (compile("sp", eqroutine); stack2[k] ← addr-1; stack1[k] ← addr-2;
[8a10]                tmp ← addrpart[addr-2]; mem[addr-2] ← "sp"; return address of this call of eqsub;
                      return (tmp) )                                  garbage
12a10         resetdvexp; go to next;
18a9                  resetdvexp = (eflag ← false; op ← "mr"; op1 ← "dv")
14a10 comma:  compile("ad", "temp"); compile("ts", l);
18a10         addrpart[stack1[0]] ← stack2[0];
22a10         resetdvexp; ~~rflag ← false~~ nflag ← false; go to next;
```

⇐ means double word
⊕ means floating add etc
all done interpretively

What we have seen so far gives enough information to see how the formula "a = b," is compiled. [Except compile subroutine appears below]

Scanned symbol	actions here ⓙ = loc of jth compiled instruction.				
a	mem[switchtable] ← sp ①	mem[①] ← mr a			
=	mem[②] ← sp eqroutine	mem[①] ← sp mm	stack1[0] ← ①	stack2[0] ← ②	l ← a
b	mem[③] ← mr b				
,	mem[④] ← ad temp	mem[⑤] ← ts a	mem[①] ← sp ②		

So the compiled code is:

```
jump to *+1
jump to eqroutine
multiply by b
add temp
store in a
```

these are interpreted at run time to refer to floating pt accumulator
i.e. flac ⇐ flac ⊗ b; flac ⇐ flac ⊕ temp; a ⇐ flac

The eqroutine does just what is needed, namely "temp ⇐ 0.0; flac ⇐ 1.0; return"!
Now let's look at routines needed for slightly more complex expressions $a = 3.1b - c/d^{-2}$ etc.:

```
0a5   number:   while rdigits > 0 do (flac ⇐ flac ⊘ 10.0; rdigits ← rdigits - 1);
4a5             j ← alloc; mem[j], mem[j+1] ⇐ flac;
9a5             compile(op, j); go to reset;
0a6   exponent: tmp ← addrpart[addr-1]; mem[addr-1] ← ("sp", exproutine);
4a6             compile(0, tmp); compile(0, decode); go to reset;
52a4  plus:     resetdvexp; compile("sp", plroutine); go to next;
55a4  minus:    resetdvexp; compile("sp", mnroutine); go to next;
0a9   divide:   op ↔ op1;
                eflag ← not(eflag); go to next;
29a4  expminus: eflag ← not(eflag); rflag ← true; go to next;
```

tricky use of rJ here, MIX equivalents is:

PLUS	MINUS	1H
JMP 1F JMP PROUTINE	JMP 1F JMP MROUTINE	STJ *+2 JMP ROSSETTA LDA * JMP COMPILE JMP NEXT

Here are the subroutines for compilation and storage allocation:

```
9a8     compile(x,y) = (mem[addr] ← (x,y);
11a8                    addr ← addr + 1;
12a8                    if addr > caddr then errorstop);
0a8     alloc = (caddr ← caddr - 2; if addr
4a8                     if addr > caddr then errorstop;
6a8                     return(caddr) )
```

Here are the floating-point routines used at run time:

```
0a13    eqroutine = (temp ⇐ 0.0; flac ⇐ 1.0)
5a13    plroutine = (temp ⇐ temp ⊕ flac; flac ⇐ 1.0)
10a13   mnroutine = (temp ⇐ temp ⊕ flac; flac ⇐ -1.0)
15a13   exproutine(x,n): (if n ≥ 0 then pwrflag ← false else (pwrflag ← true; n ← -n);
29a13                    while n > 0 do (if pwrflg then flac ⇐ flac ⊘ x else flac ⇐ flac ⊗ x; n ← n-1))
```

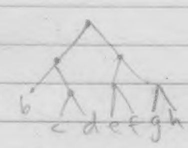
Note bug: a = -b comes out the same as a = 1-b. [See Laning's letter.]

Now, parentheses:

```
0a11  lpren:  compile (0,0);
1a11          k ← k+1;
3a11          eqsub;
4a11          stack3[k] ← alloc;
7a11          if op < op1 then go to next;        enumerl: "mr" < "dv";
11a11         stack1[k] ← stack1[k] + 2^11;
12a10         resetdvexp; go to next;
0a12  rpren:  j ← stack1[k] mod 2^11;
6a12          if stack1[k] < 2^11 then resetdvexp
28a9                else (eflag ← true; op ← "dv"; op1 ← "mr");
9a12          compile ("ad", temp); compile ("ts", stack3[k]);
19a12         stack2[k] ↔ stack2[k-1]; compile ("sp", stack2[k]);
28a12         addrpart[j] ← addr;
30a12         compile (op, stack3[k]); k ← k-1; go to next;
```

Interpretation is stack3[k] is location to hold contents of this parenthesis pair
stack1[k] is location of instruction "sp ——" just preceding the code for this parenthesis pair, it will later jump to an instruction that uses the contents of the parenthesis after evaluation [exc k = 0 ⇒ it will evaluate the exp]
stack2[k] is location of instruction to begin the evaluation of the parenthesized expression.

Example a = ((b(cd))((ef)(gh))),



leads to:

Scanned symbol (1) (13) (2) (14) (3) (15) (4) (16) (5) (17) (6) (18) (7) (19) (8) (20) (9) (21) (10) (22) (11) (23) (12) (24) stack1 stack2 stack3

Scanned symbol	code generated	stack1	stack2	stack3
a	mr a			
=	sp • sp (=)	(1)	(2)	
(	sp • sp (=)	(1)(3)	(2)(4)	[1]
(	sp • sp (=)	(1)(3)(5)	(2)(4)(6)	[1][2]
b	mr b			
(	sp • sp (=)	(1)(3)(5)(8)	(2)(4)(6)(9)	[1][2][3]
c	mr c			
d	mr d			
)	ad tmp ts [3] sp (6) mr [3] sp (15)	(1)(3)(5)	(2)(4)(9)	[1][2]
)	ad tmp ts [2] sp (4) mr [2] sp (19)	(1)(3)	(2)(9)	[1]
(	sp • sp (=)	(1)(3)(20)	(2)(9)(21)	[1][4]
(	sp • sp (=)	(1)(3)(20)(22)	(2)(9)(21)(23)	[1][4][5]
e	mr e			
f				
)		(1)(3)(20)	(2)(9)(23)	[1][4]
(		(1)(3)(20)(30)	(2)(9)(23)(31)	[1][4][6]
g				
h				
)		(1)(3)(20)	(2)(9)(31)	[1][4]
)		(1)(3)	(2)(31)	[1]
)		(1)	(31)	
,				

Final code:

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
	sp (31)	sp (=)	sp (45)	sp (=)	sp (19)	sp (=)	mr b	sp (15)	sp (=)	mr c
10+:	mr d	ad tmp	ts [3]	sp (6)	mr [3]	ad tmp	ts [2]	sp (4)	mr [2]	sp (41)
20+:	sp (=)	sp (29)	sp (=)	mr e	mr f	ad tmp	ts [5]	sp (21)	mr [5]	sp (31)
30+	sp (=)	mr g	mr h	ad tmp	ts [6]	sp (23)	mr [6]	ad tmp	ts [4]	sp (9)
40	mr [4]	ad tmp	ts [1]	sp (2)	mr [1]	ad tmp	ts a			

which is equiv to:
1×g×h+0 → [6]
1×e×f+0 → [5]
1×[5]×[6]+0 → [4]
1×c×d+0 → [3]
1×b×[3]+0 → [2]
1×[2]×[4]+0 → [1]
1×[1]+0 → a

Finally, control flow:

```
42  print1:  lowercasecode[";"] ← next; lowercasecode["P"] ← print2; lowercasecode["."] ← period; go to label;
2a2 print2:  compile("sp", pntroutine);
             for j ← 1 until 4 do trash ← read next char;   comment R, I, N, and T;
7a2          go to next;
42  period:  lowercasecode[","] ← comma; lowercasecode["P"] ← print1; lowercasecode["."] ← 0; nflag ← false; go to next;

40a10 ucs:   scop ← "sp"; go to ucsc;   comment SP and CP can't be labeled;
50a10 ucc:   scop ← "cp";
42a10 ucsc:  c ← read next char; if c = "R" then go to 0; comment "provision for future SR, CR instructions";
45a10        if c = "T" then go to 32; comment "START the object program;
47a10        lowercasecode[","] ← spcomma1; go to next;
53a10 spcomma1: lowercasecode[","] ← spcomma2; compile(scop, switchtable + decode); go to reset;
54a10 spcomma2: lowercasecode[","] ← comma; go to next;
```

Runtime print routine:

```
0a15  pntroutine: "si 149"; j ← location of first parameter;
3a15    while oppart[j] ≠ "sp" do (print character(addrpart[j]); print character("=");
                                   floating point print(mem[j]); j ← j+1);
7a15    go to j;
42a4  upper: cflag ← true; go to next;
0a4   lower: cflag ← false; go to next
```

Total length of compiler (incl 15 for stack storage) $45+20+64+64+19+26+6+19+34+78+19+38+13$ words $= 445$

plus runtime routines $43+19$ $= 62$.

object program goes into locs 32-196; 210-221 is switch table
subscripted variables, drum overlay usage, D operator: not yet implemented.